Esmeralda by Example

Lars Schillingmann

December 2006

Esmeralda is a collection of tools dealing with the tasks needed to build an automated speech recognition system. Esmeralda supports e. g. feature extraction using MFCCs, PCA, vector quantization, classification with mixture densities, initialization, training and alignment of HMMs, adapting HMMs using MLLR and MAP, stochastic language models (n-gram models) and grammars. The incremental speech recognizer (ISR) uses many of these components to generate its hypotheses. Due to its modular design Esmeralda can also be used in a wide range of other pattern recognition tasks. This tutorial covers a short introduction to Esmeralda.

1 Introduction

Please note, that the author is not a developer of Esmeralda[1]. Therefore his knowledge about Esmeralda is partial and this tutorial may contain errors. You know who to blame[2].
The idea behind this tutorial is to guide you through the process of building a small speech recognition system using Esmeralda. After this you will have learned the basic usage of most components. This tutorial assumes you are familiar with *nix and typical command line tools such as cut, paste, grep and sed. You also should have fun when working with large chunks of ASCII data.

1.1 Installation Notes

Esmeralda is written in C and is known to successfully compile under Solaris, OSF1, SuSE and Gentoo (authors experience).

[1] Environment for Statistial Model Estimation and Recognition on Arbitrary Linear Data Arrays
[2] lschilli@techfak.uni-bielefeld.de

Some quick notes about installing:

- You need to set an environment variable `ESMERALDA` pointing to the directory with the ESMERALDA directory structure containing src, include, bin, lib, man.
- You need to set an environment variable `ARCH` to your OS-architecture e. g. linux or sol2. See `Makerules`.
- You certainly need to modify `Makerules` in the `src` directory and probably the `Makefile` in the `src/isr` directory e. g. to exclude the DACS communication framework.
- The build process does not stop on errors so keep an eye on the log.
- The build process seems to run into some endless recursive loop on some systems, but successfully builds all binaries. This happens if non-existing modules are listed in the makefile.
- You should directly run `make install` instead of `make` because libraries required by further build steps are expected to be installed in `lib`.
- Check if mm/lib successfully compiles when using Linux. This component needs flex and yacc.
- Although recursive make is a traditional method building projects the author suggests not using this for your own project[3].

1.2 Environment

The arguments passed to ESMERALDA tools on the command line can be quite complex and are partially order dependent. This leads to problems when working with Linux, as the bash seems to resort options on the command line there. Setting `POSIXLY_CORRECT=1` enables the POSIX mode and so prevents resorting.
The makefile which prepares files for the isr we will meet later in this tutorial is sensitive to the language depended behaviour of sort. Set `LC_COLLATE=POSIX` if you experience problems.

1.3 Overview of Binaries

Table 1 on the facing page gives an overview of the binaries available within ESMERALDA. The abilities listed there are not complete but should give a rough overview. As one can see the idea behind ESMERALDA is to divide the functionality into several small tools.

[3]Recursive Make Considered Harmful: `http://aegis.sourceforge.net/auug97.pdf`

	es_classify es_create es_project	classification using eigenspaces
feature extraction	dsp_fex	feature extraction (MFCCs)
	dsp_vad	voice activity detection
	fx_deriv	derives features
	fx_extract	extracts parts of feature vector sequences
	fx_norm	normalizes feature vectors
	fx_select	selects components from feature vectors
	fx_stat	calculates statistics per dimension (min, max, mean, σ)
mixture densities	md_filter	transforms feature vectors
	md_init	creates a new codelibrary from labeled feature vectors
	md_k_means	creates a new codelibrary from unlabeled feature vectors using k-means
	md_lbg	creates a new codelibrary from unlabeled feature vectors using LBG
	md_train	trains a codelibrary
	md_param	modifies/updates a codelibrary with accumulator data
	md_classify	classifies with mixture densities
markov models	mm_init	creates a new HMM (states, models) using an initial annotation
	mm_train	trains a HMM using viterbi or baum-welch
	mm_param	modifies/updates a HMM with accumulator data
	mm_align	performs an alignment with a HMM
	mm_adapt	adapts a HMM using MLLR
	mm_count	counts occurrances of named concepts
	mm_pmod	creates concept definitions from phonetic transcriptions
	mm_select	select concepts from a HMM
	mm_swu	creates sub-word units for the isr
	mm_tree	generates a state tree for the isr
lang. model	lm_check	checks an n-gram model for consistency
	lm_count	generates n-gram count data
	lm_param	creates n-gram parameters
	lm_perp	calculates perplexity
	grm_tab	generates effective lexicon and parse table from grammar
	isr	the incremental speech recognizer
	ev_det ev_seg	evaluation of segmentation results

Table 1: Overview of binaries

1.4 Getting Help

All ESMERALDA binaries give an option listing when called with `-h`. Some of them have a modular design. Available modules are listed within square brackets:

```
usage: isr [<option> ...] <swu> <lex> <tree> [LM] [CL] [AA] [FE] [SR]
```

Calling `isr` with `-h` will print help for the isr main module but not for e. g. the LM module. To get help for that module you need to provide all mandatory options for the modules before:

```
isr - - - -h
language model usage: isr ... [-g <grm>] [<option> ...] <lm>
...
```

As one can see this will only work if the command line does not get resorted. On problems see section 1.2 on page 2.
Unfortunately that's nearly all you can get. There are some manpages existing especially for some of the file formats the tools read and write this may be interesting:

```
man/man1:
md_filter.1  md_init.1  md_k_means.1  mm_align.1  mm_init.1
mm_param.1   mm_train.1

man/man4:
grm_def.4  md_codebook.4  mm_concept.4  mm_model.4  mm_state.4
```

1.5 Locations within the File System

There might be some pathnames which are good to know:

- The current location of the stage directories needed for this tutorial is:
  `/vol/speech/britta/ASR_Uebungen` (Techfak-Unix)
- This tutorial uses signal data available under:
  `/vol/sprachdaten/ssg/Zahlen` (Techfak-Unix)
- The most current ESMERALDA binaries and sources are installed under:
  `/coda/vol/esmeralda/` (M5-Linux)
- The makefile to generate the `isr` specific formats used in this tutorial is a modified copy from here:
  `/vol/mobirob/BIRON_SunShine_from_CVS/SpeechRec/cfg/NOT_IN_CVS/`
  (M5-Linux)
- There are some modules which do not seem to belong officially to ESMERALDA although they are mentioned in the makefile. They are located at:
  `/coda/homes/tploetz/src/esmeralda/src/` (M5-Linux)
  The ev module which contains `ev_seg` and `EvalSimple` is used in this tutorial.

1.6 Feature Extraction

In the next chapter we assume features are already extracted. If you want to extract features for speech recognition purposes there is one tool available within ESMERALDA called `dsp_fex`. This tool is quite simple to use. It operates in two modes. One takes an input file (raw, 16 kHz, mono, signed word) and the output file as arguments. The other one batch processes a list of source destination filename pairs. Available feature extraction types are MFCCs in several versions. For further information take a look at the source file `mfcc.c` in `dsp/lib`.

2 Stages

Prepared with the most basic info you are now ready for the main part of the tutorial. Each subsection in this chapter has a companion piece in the supplied tutorial directory structure prefixed with stage. Due to the number of files which need to be created and the number of tools involved it is easy to get lost. Using stages is an attempt to avoid this. In each stage there are symlinks to files which need to be processed or used in this stage. Symlinks in further stages refer to files in earlier ones if they are needed there. Dependencies are more explicit and easier to understand like that. Figure 1 shows the stage structure. A gray arrow denotes that symlinks from this stage to files within the other stage exist.

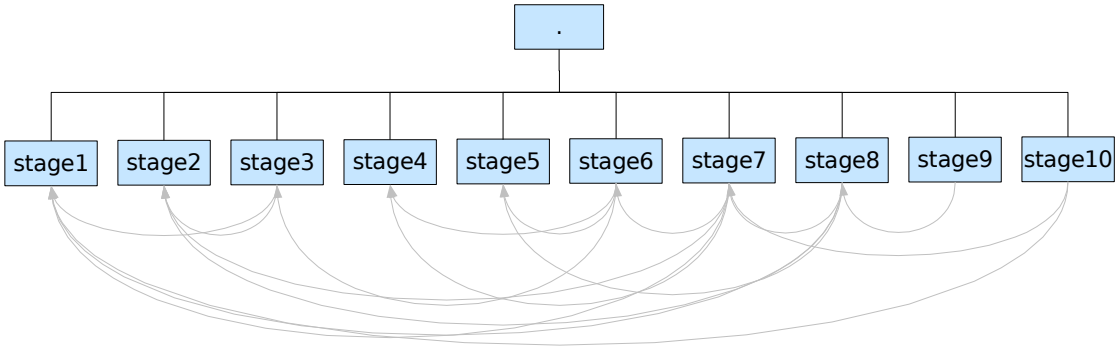

Figure 1: The stage structure and dependencies

Furthermore in every stage directory there will be some shell scripts. Normally they are named `DoCreate*.sh` and if so they will not always be mentioned explicitly in the according section. These scripts will automatically perform the steps described in the tutorial to ensure you will not miss any details probably not mentioned in this tutorial. In most cases you only need to execute one script. For a first walkthrough stick to the ones which contain 'sil' in their filenames. You should always read these scripts. Do not run the ones containing '3ph' (see section 3.1 on page 18).
While reading this tutorial you will notice that filenames are not mentioned explicitly but only their file extensions which are of course not mandatory. Besides the lack of complete filenames does not make the text ambiguous the idea behind this is to make some already existing naming conventions more explicit and just name the concept behind it — while keeping a strong link to the physical file — rather than the complete filename.

2.1 Stage 1 – Phonetic Transcriptions

The next few stages will deal with the steps needed to perform a so called *forced alignment*. Basically this is using an existing HMM to create an alignment with training data. This is used as an initial annotation to create a new HMM. To perform that step we need to make some preparations first.
We want to use phonetic models so we need to map words to their phonetic transcription.

These mappings are called subword unit definitions (`.swu.def`). We will use a phonetic transcription and `mm_pmod` to convert it into a format with distinguishable model names.

Figure 2: `mm_pmod` I/O diagram

```
vierzehn        fi6|tse:n
vierzehn %=     /f/ /i/ /6/ /t/ /s/ /e:/ /n/ ;
```

Note that we need to exchange the := operator with the %= operator. Otherwise `mm_align`, we encounter later, will not generate hypotheses on phoneme level, but on word level. See `man mm_concept` for further details.
In addition to that we have to introduce a silence concept:

```
<sil> %= { <-> | <--> | <---> } ;
```

The | character means 'or'. The 'phonemes' between the angle brackets refer to silence models of different lengths. See `DoCreate.sil.sh` for the complete call.

2.2 Stage 2 – Corpus Creation

In this stage we will create a corpus. This is a driving file telling `mm_align` (next stage) which concepts exactly it should align on what data. We will suffix this file with `.Corpus.train` because it will contain the training data filenames and the correspondent concepts. We also want to align the previously defined silence concept optionally at the beginning and the end of the data. This is denoted by the question mark.

```
/.../Zahlen/ufv/aa02/Z0000   <sil>? null <sil>? ;
/.../Zahlen/ufv/aa02/Z0001   <sil>? eins <sil>? ;
```

The data files contain a sequence of feature vectors which have to be generated using `dsp_fex`. They are often suffixed with `.ufv` which means unified feature vectors.
In our example the mapping between filenames and numbers is done by a small tool in the stage2 directory named `hdr2corpus.sil.py` which uses a filename to number mapping to generate the `Corpus.train` file from a list of filenames. See `DoCreate.sil.sh` for the complete call.

2.3 Stage 3 – Forced Alignment

In this stage we will use the previously mentioned `mm_align` to generate an alignment using the files from the preceding steps. To understand which data `mm_align` expects we need to explain how HMMs are modelled in Esmeralda.

2.3.1 Codelibraries

ESMERALDA supports semi-continuous HMMs which means multiple states share the same mixture densities but weighting them differently. Therefore it makes sense to store the mixture densities separated from the state definitions. That is where codelibraries come into. A codelibrary contains mixture density definitions (multiple codebooks) which means it contains mean vectors and covariance matrices. In our case the codelibrary contains only one codebook and therefore defines one mixture density. Codelibrary files are suffixed with `.cl`. Sample excerpt:

```
# definition for class # 0
# class name, prior probability
''      0.000898134
# class type is GaussDiag
# mean vector
-3.91937 -0.496206 0.124862 -0.124245 0.140082 -0.224351...
```

For further details see `man md_codebook`.

2.3.2 States

Since a HMM needs to be represented somehow the states must be defined. This is done in a `.state` file. A `.state` file generally contains state numbers, transition probabilities, emission probabilities, the used codebook and the weights for each density in the codebook.

```
0       2:[0.977678 0.0223218 ] 0/1024:[0.00054486 0.000584609...
1       2:[0.972308 0.0276916 ] 0/1024:[0.00372407 0.00485151...
```

The numbers within the first square brackets denote the transition probabilities. There are only two in this case because they belong to a linear model and therefore we have only two possible transitions from each state. See `man mm_state` for further details.

2.3.3 Models

We still have no complete HMM. Multiple states need to be put together to form a HMM. This is done in the `.model` file. It contains model names, the model topology and the state numbers the model consists of. See `man mm_state` for further details. Sample excerpt:

```
/a/     Linear  2:[10 11 ]
/a:/    Linear  3:[12 13 14 ]
/e/     Linear  2:[15 16 ]
/e:/    Linear  3:[17 18 19 ]
```

As one can see in this case each HMM models a phoneme. To align words we actually need to model words as HMMs. We already did this:

The word to phonetic transcription mapping done in the `.swu.def` (see stage 1) should not only be seen as just a mapping but also as definitions of bigger word HMMs consisting of (in this case) phoneme HMMs. These are called compound models.

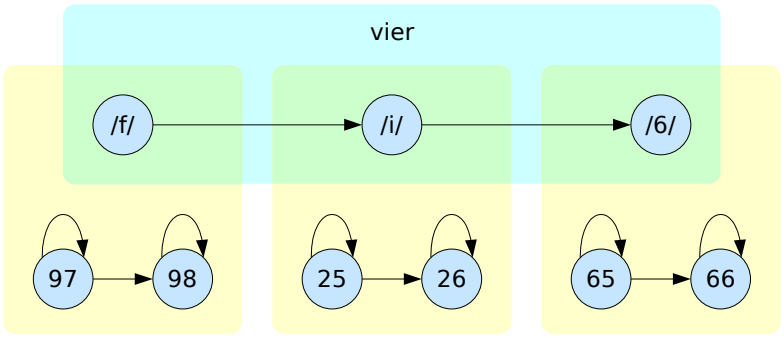

Figure 3: compound model

2.3.4 Putting It All Together

Remembering the previous steps you may have noticed that we did not create a codelibrary, a state file or a model file. We did not because without any initial annotation we cannot initialize a HMM. One possibility to get one automatically is to use a forced alignment. This method needs some existing HMM which is ideally trained with somehow similar data.
In this case we use an ideal candidate which contains phoneme models already trained on other data. The alignment is called forced because the corpus from step 2 enforces a specific word model to be aligned for each sequence of feature vectors regardless if other models fit better. As a result we will get an alignment on phoneme level due to the definitions in the `.swu.def` file.
The I/O diagram in figure 4 gives an overview of the files involved when using `mm_align`. In our case the result contains something like:

```
<->[0..40] /f/[41..50] /i/[51..64] /6/[65..72] /t/[73..79]...
?
<->[0..46] /z/[47..48] /E/[49..58] /C/[59..67] /t/[68..76]...
<-->[0..4] /z/[5..14] /i:/[15..20] /p/[21..40] /t/[41..43]...
```

The numbers in the square brackets denote the range of feature vectors aligned to one model. A question mark denotes if the models cannot be aligned on a feature file with a production probability above some threshold.
Since we want to initialize a HMM using such an alignment in one of the next steps, we need to connect the alignment with their respective filenames and filter out any unsuccessful alignments. This can be done with `paste` and `grep`. The resulting file is suffixed with `.Corpus.ini` – since it will be the corpus for HMM initialization. See `DoAlign.sh` for the complete call.

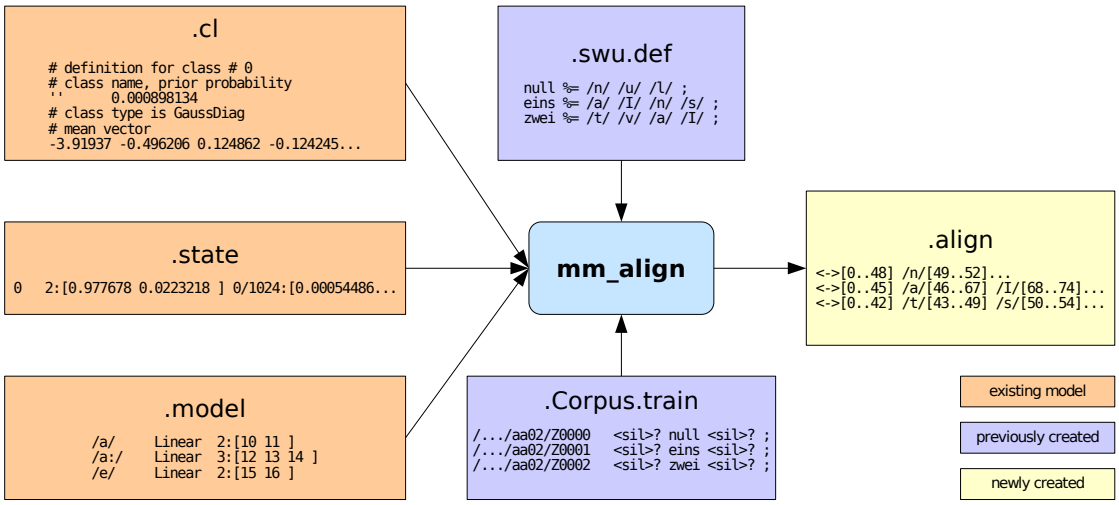

Figure 4: `mm_align` I/O diagram

2.4 Stage 4 – Codelibrary Creation

Before we can initialize a HMM in one of the next steps we need to estimate mixture densities on our training data. In ESMERALDA terms this means creating a codelibrary. We will use the k-means algorithm which is implemented by `md_k_means`.
Calling `md_k_means` is quite simple. It just takes the feature vector dimension, the number of densities (classes) to be created and a list of feature files (called .rcs – for resources) to read. The result is a codelibrary (see section 2.3.1 on page 8). The `md_k_means` I/O diagram is shown in figure 5. The script in the corresponding stage directory makes use of the additional parameters `-p` to get better initialization vectors and `-g` due to the small amount of data available. See `man md_k_means` for details.

Figure 5: `md_k_means` I/O diagram

2.5 Stage 5 – Phonemes Only

In the last step before initializing the HMM we will create a new subword unit definition file to define a compound model named `PHONES`. This compound model should contain all phonemes occurring in the phoneme level alignment described in 2.3.4 on the preceding page. It should allow arbitrary sequences of the phonemes. We will suffix it with `.phones.def`.

```
PHONES %= {
<-> |
```

```
<--> |
<---> |
/6/ |
/9/ |
...
/z/ }+ ;
```

`mm_init` will not directly need this in the next step but *we* will need it here to have an easy way telling `mm_init` what models of which kind it should create. The file generated here will then also be used in further steps for evaluation purposes.

2.6 Stage 6 – HMM Initialization

In this stage we will use `mm_init` to initialize a new HMM. Before we begin we have to tell `mm_init` which models and model types to generate. We will do this by creating a so called `.model_frame` file. In our case will use `egrep` and `sed` to automatically create it from `.phones.def` file. This result has the following format:

```
<->     Linear
<-->    Linear
<--->   Linear
/6/     Linear
/9/     Linear
...
```

It contains just a list of the subword units each followed by the model type. In our case we use a linear model. There are other types like e. g. Bakis. See `man mm_model` for additional information.
`mm_init` requires a `.model_frame` and the initial annotation `.Corpus.ini` created in section 2.3.4 on page 9. We will also provide the previously created codelibrary (see section 2.4 on the facing page). In our case `mm_init` will create an initial `.state`, a `.model` and a `.stats` file (cp. section 2.3.2 on page 8 and 2.3.3 on page 8). The last file contains statistical data such as the average length of a model and is not needed by further steps. Figure 6 on the next page gives an overview.
Please note that `mm_init` will not perform any training steps. (See next stage for that.) It will just create an initial HMM by basically doing some counting. For further details see `man mm_init`. To get more information about the model creation options you may also read `mm_init - - - -h`.

2.7 Stage 7 – HMM Training

ESMERALDA performs HMM training in two steps. Firstly `mm_train` does one reestimation step (Baum-Welch or Viterbi) the results are so called accumulator data. In our example two files are created. An `.update` and an `.accu` file. Secondly these files will be used to update the HMM. The `.update` file is post processed by `md_param` in order to create an updated codelibrary (`.cl`) and the `.accu` file is post processed by `mm_param`

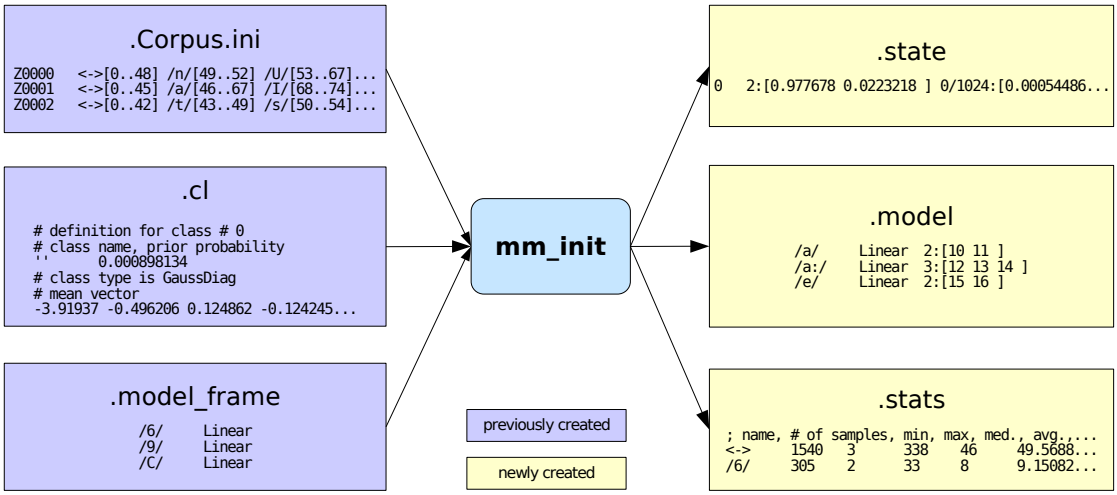

Figure 6: `mm_init` I/O diagram

to create an updated `.model` and `.state` file. In our case the model file will not change since we do not use generalisation hierarchies. For further details see `man mm_param` and `md_param -h`.

Since we have no exact phoneme based annotation of our training data we will train the HMM word based. We already have created a word based annotation (`Corpus.train`) in stage 2 (see section 2.2 on page 7). The subword unit definitions (`.swu.def`) from stage 1 (section 2.1 on page 6) are also needed as we have to define the word HMMs as compound models of the phoneme models we have previously defined (cp. the `.model_frame`).

Of course `mm_train` also needs the HMM initialized in the previous step, i. e. the `.model` and the `.state` file. The codelibrary from stage 4 (section 2.4 on page 10) is needed again. All in all in our example `mm_train` will read five files and write two. Figure 7 on the next page gives an overview. Figures 8 and 9 illustrate the post processing.

In order to perform multiple training iterations running these tools is not a trivial task. Therefore the `DoTrain.sh` script automates the training. `DoTrain.sh` takes two arguments: One for the iteration to begin with and one for the iteration to stop after. `DoTrain.sh` does not overwrite files but creates new ones in like `.1.state`, `.2.state`, ... — one for each iteration.

After we have trained our HMM we have to think about some evaluation to see if it works. The `DoTrain.sh` script generates `.accu.err` files which contain the `mm_train` output for each training iteration. To get a basic idea if the training process converges we can take a look at the average $P(O|\lambda)$ `mm_train` writes out. Keep in mind that this is a negative log probability when interpreting this value. To get a more exhaustive idea about the quality of our HMM we have to do some real evaluation, i. e. use it with `mm_align` on training data and analyse the results. But in the first place we have to do some preliminaries again.

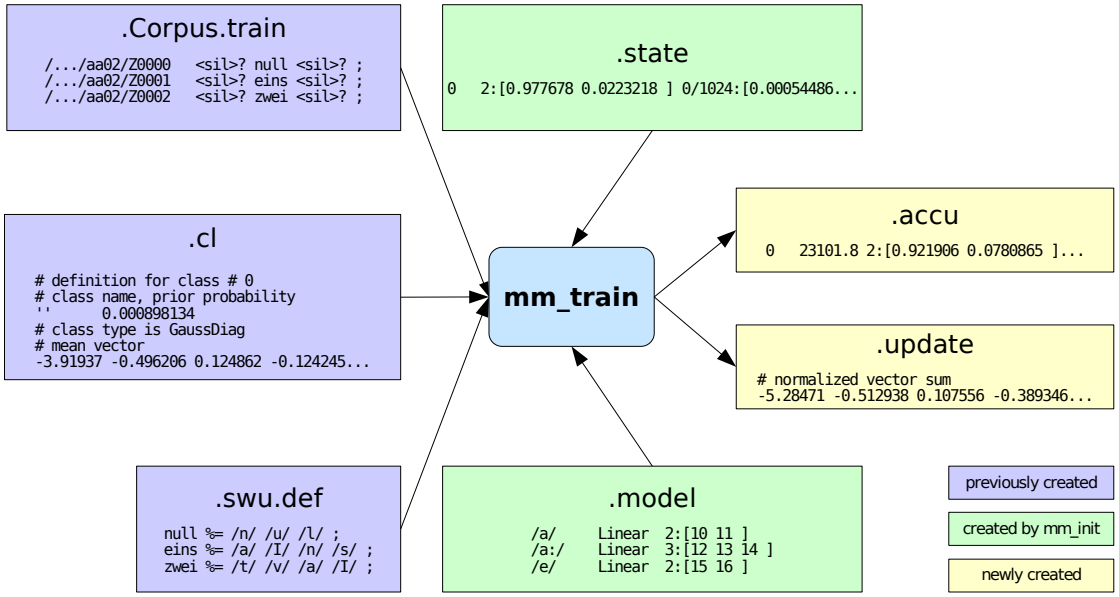

Figure 7: `mm_train` I/O diagram

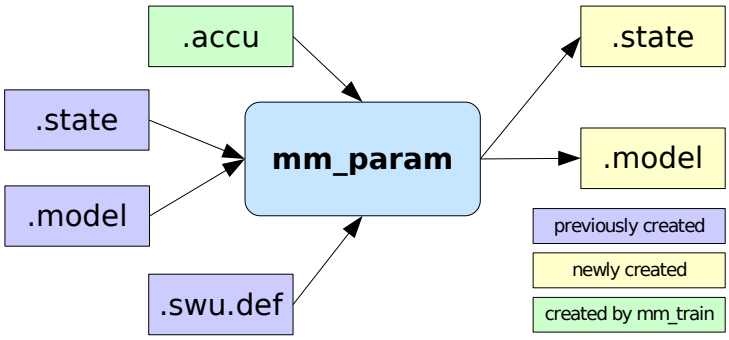

Figure 8: `mm_param` I/O diagram

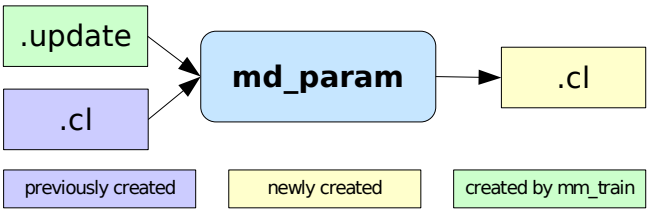

Figure 9: `md_param` I/O diagram

2.8 Stage 8 – Concept and Corpus Creation

In the previous steps we have created different subword unit definition files. One contains phonetic transcriptions of our lexicon (see 2.1 on page 6), the other one contains a `PHONES` concept (see 2.5 on page 10). We want to use `mm_align` in the next step to compute a word level alignment of our testing data. But the currently available concepts do not support something like letting `mm_align` choose which word to align. To achieve this we just have to define another HMM structure. By defining a `LEX` concept which contains all word concepts and allows any order and number of words we will be able to get our desired alignment. This also requires the word concept definition to be present in that file. Since we additionally want to perform a phoneme level alignment we will also merge the phoneme concept in that file. This file is suffixed `.lex.def`.
When looking at the files you may have noticed a slight difference between the word concepts in this file and the one from stage 1:

```
null %= /n/ /U/ /l/ ; (stage 1)
null := /n/ /U/ /l/ ; (this stage)
```

In stage 1 we used the `%=` operator because we wanted a phoneme level alignment when using word concepts. In this stage we do not want individual hypotheses for each phoneme when we use the `LEX` concept which refers to the word concepts. This is the difference between the `%=` and the `:=` operator. (cp. `man mm_concept`)
In order to use the `PHONES` and the `LEX` concept in conjunction with our testing data we have to create the appropriate corpora. We will suffix the corpus file using the `PHONES` concept `.Corpus.phones.test` and the file using the `LEX` concept is suffixed `.Corpus.test`.
Everything should be prepared to run `mm_align` now.

2.9 Stage 9 – Evaluation

Everything is prepared: We just have to run `mm_align`. The `DoEval.sh` script will create a word level alignment and the `DoEval.phones.sh` will create the phoneme level alignment. They take one argument to define the training iteration to use the HMM from. `mm_align`s usage is explained in section 2.3 on page 7. The script also will merge both the correct annotation and the hypothesis to an `.eval` file. This can be further processed by `EvalSimple`. `EvalSimple` calculates some measures which are useful to estimate the recognition performance of the HMM. Since the output is a little cryptic, here is an overview:

WA	Word Accuracy (1 - Word Error Rate)
SR	Sentence Recognition Rate
WC	Word Correctness
S	Substitutions
D	Deletions
I	Insertions

The phoneme level alignment can be used by `DoHyp2TextGrid.sh` to create so called text grids. These files can be read by `praat`[4] to make the alignment audible and visible as segmentation of the source audio file.
Actually if the results are good we now have a working speech recognition system. We can recognize words, in our case numbers. But this is probably not what you imagine when talking about a speech recognition system, since it is neither working online nor able to process audio data directly. To solve this we will prepare the HMM to be used by the incremental speech recognizer.

2.10 Stage 10 – The incremental speech recognizer (isr)

As the title says the `isr` is incremental and designed to real-time process speech. Therefore the `isr` needs HMMs in another format than the `.state` and `.model` format (cp. section 2.3.2 on page 8) in order to fulfil this requirement efficiently. The HMM now has to be represented as a `.swu` and a `.tree` file.
The `.swu` is in principle a fusion between a `.model` and a `.state` file:

```
/a/     0       0.802271/0.197729       0:[0 0 0...
```

The codelibrary (`.cl`) is still needed as before since it contains the densities.
The `.tree` file can in principle be understood as a tree representation of the `.swu.def` file, so that after walking through this tree the probability at the last state decides which word has been recognized.
Since the creation of these files is rather complicated and involves a bunch of tools we will use an approach using a makefile which automates these steps almost completely. The makefile processes the `.state`, the `.model`, the `.swu.def` and a grammar (`.grm`) file. The results are a `.swu`, a `.tree` and a lexicon (`.lex`) file. There are more files generated but they seem to serve as temporary files or have a function the author is currently not aware of. The lexicon defines the words the `isr` is able to recognize. The grammar definition has an important role in this context. Besides the `isr` can use it as a language model to influence and restrict the recognition process the grammar is also being used as a source to generate the lexicon from. So you just need to provide the grammar and HMM and the makefile does the rest. Below is an example of a very simple grammar:

```
$$S: $Zahlen ;

$Zahlen:
null |
eins |
...
hundert ;
```

[4] `http://www.praat.org`

The $-character marks symbols as non-terminal. Symbols with two $-characters denote so called technical symbols which will not be outputted as hypothesis. To learn more about the grammar definition format see `man grm_def`. Figure 10 shows which files are read and created by the makefile.

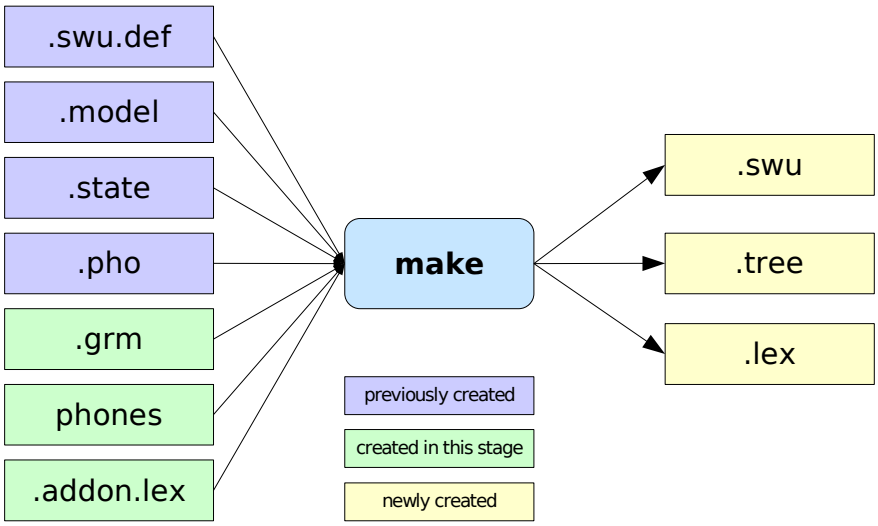

Figure 10: `isr` makefile I/O diagram

You may have noticed three unmentioned files: `.pho`, `phones`, `.addon.lex`. The makefile is able to determine words not available in the current concept (`.swu.def`). Concepts for these words are automatically added using `mm_pmod` (cp. section 2.1 on page 6), their transcription (`.pho`) and a phoneme list (`phones`). The `.addon.lex` file defines concepts which should be recognized but ignored like e. g. <breathe>. Note that the silence concept <sil> is expected and should not be present in that file.
After creating the necessary files we can now run the `isr`. Since the `isr` is probably one of the most complex command line programs the `isr.sh` will give you a (hopefully) working example. The `isr` consists of multiple modules whereas each has its module specific mandatory and optional arguments and switches. In case of problems be sure to read the help for the correct module and you obeyed section 1.2 on page 2 and section 1.4 on page 4. `isr.sh` expects one argument: The input file, which can e. g. be `/dev/dsp` or a raw audio file (16 kHz, mono, signed word). Figure 11 on the facing page shows the `isr` I/O regarding our example.

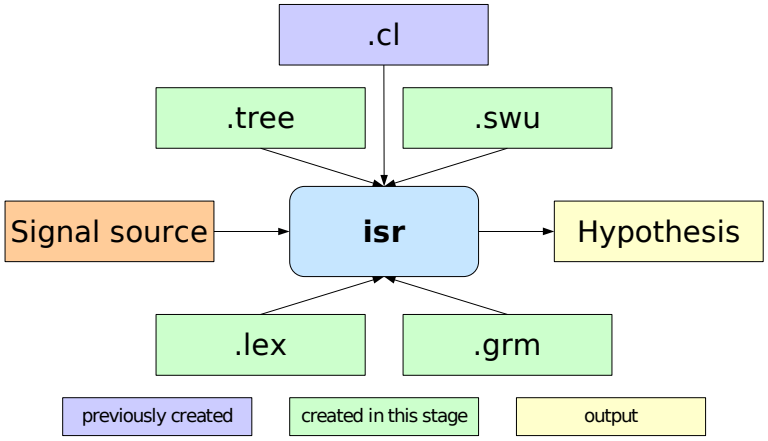

Figure 11: `isr` I/O diagram

Below is an example of an `isr` output:

```
what / (hello ) (that's it ) what can I
what / (hello ) (that's it ) what can I do / ;
your ? ;
```

This output requires some explanation: Each text within brackets is a recognized phrase. The slash means the parsing process of the grammar has been cancelled since the next word does not match it any more. The '?' denotes a word the grammar does not match.

That's it. You have just built your first simple speech recognition system using ESMERALDA! If you test it you need to talk a little (up to one minute) to it before the `isr` has adapted to the signal channel properties. These channel definition parameters can also be saved for later usage. See `isr - - - - - - -h` for help on that.

3 Advanced stuff

This section contains short notes on other things which can be done with this tutorial or with ESMERALDA.

3.1 Triphonemes

You may have noticed the `Do*_3ph.sh` in some of the stage directories. This variant creates the files needed to perform this tutorial using triphonemes. You will notice another `.swu.def` which additionally contains lines like:

```
#/E/l <= /E/ ;
#/a/I <= /a/ ;
#/a/x <= /a/ ;
```

This is called a triphoneme to monophoneme mapping and builds a generalization hierarchy. While training HMMs are created for both triphonemes and monophoneme. The monophonemes receive the all the training data of their corresponding triphonemes. Normally there are multiple contexts for one phoneme so a monophoneme receives more data and has therefore probably more evidence than a triphoneme. The corresponding monophoneme can automatically be used if the triphoneme did not gain enough data.

3.2 PCA with Esmeralda

Some quick notes about doing a PCA with ESMERALDA:

- Make sure your features are normalized. If not, use `fx_stat` to calculate a normalization table.
- Use `md_init -u -T [.pca] [input_dim] [.ufv.rcs] \ -d [output_dim] PCA >[.cl]` to create the transformation matrix.
- Use `md_filter [input_dim] [.pca] <[in.ufv] >[out.ufv]` to transform feature files.

3.3 Additional Stages

There are some additional stages available:

Stage 11	contains some scripts to record numbers on your own. They have some characteristics to enable the use of a language model.
Stage 12	actually creates these models using `lm_count`.
Stage 13	uses the `isr` in conjunction with these models.

4 Acknowledgments

The author would like to thank Britta Wrede and Sven Wachsmuth for many explanations. The stages this howto is based upon have been created by Britta Wrede and the author to serve as exercises for the speech recognition course during summer 2006.